УДК 519.245; 519.674; 004.032.24:004.272

П.О. Приставка, А.К. Шевченко

Національний авіаційний університет

ДОСЛІДЖЕННЯ РЕАЛІЗАЦІЇ ЛІНІЙНОГО ОПЕРАТОРА ЗГОРТКИ ЦИФРОВОГО ЗОБРАЖЕННЯ ПРИ 16-БІТНИХ ОБЧИСЛЕННЯХ

Досліджено реалізацію згортки складової растра з квадратною маскою лінійного двовимірного фільтра. Доведено, що за допомогою 16-бітних операцій та трансляції мовою асемблера можна отримувати виграш від 8 до 11 разів по швидкодії, при використанні у програмному забезпеченні.

Ключові слова: *згортка, цифрове зображення, лінійний оператор, авто-векторизація , SIMD, асемблер.*

Исследована реализация свертки составляющей растра цифрового изображения с квадратной маской линейного двумерного фильтра. Доказано, что при использовании 16-битных операций и встроенного ассемблера, можно получить выигрыш от 8 до 11 раз, по быстродействию в программном обеспечении.

Ключевык слова: *свертка, цифровое изображение, линейный оператор, авто-векторизация , SIMD, ассемблер.*

It has been researched the convolution of digital image raster component and two-dimensional linear filter square mask. It has been proved one may obtain software performance improvement from 8 to 11 times in the case of 16-bit SIMD operation and GCC Inline Assembly usage.

Keywords*: convolution, digital images, linear operator, auto-vectorization, SIMD, assembler.*

Постановка проблеми. Процедура автоматичної векторизації програмного коду, базис якої складають SIMD команди асемблера, є основою для прискорення значної кількості програм [1].

Отже, стан розвитку сучасних компіляторів можна відслідковувати за тим, як кожен із них справляється із завданням векторизації коду, та чи є після цього помилки виконання (що не є рідкістю). Сучасні спеціалісти надають цьому чиннику дуже великого значення, отже, використовують саме той компілятор, що призведе до підвищення швидкодії їх програм.

Особливо важливим є прискорення процесу обробки цифрових зображень (ЦЗ) як окремо, так і відеокадрів (при опрацюванні відеопотоку).

Тут треба враховувати три головні фактори: 1) обчислювальна складність методу обробки ЦЗ; 2) чи оптимально він (метод) був реалізований, у вигляді програмного коду; 3) та апаратні можливості.

Отже, надамо приклад: однією з найпростіших операцій при роботі з ЦЗ є згортка кольорової (або монохромної) складової растра з квадратною маскою деякого лінійного двовимірного фільтра:

$$p_{i,j} = F\left(p_{i,j}\right) = \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}\, p^{\circ}_{ii,jj} \qquad (1)$$

де: $i = \overline{-\frac{k_i}{2}, \frac{k_i}{2}}$; $j = \overline{-\frac{k_j}{2}, \frac{k_j}{2}}$; а $p_{i,j}$ – кольорова складова растра; $F\left(i,j\right)$ – лінійні фільтрації зображення; $\left(i,j\right)$ – індекс растра пікселів; k_i, k_j – розміри кадру зображення; $p^{\circ}_{ii,jj}$ – кольорова складова растра ідеального неспотвореного зображення; $\gamma_{ii-i,jj-j}$ – елемент маски фільтра; $\left(2r_i+1\right)\times\left(2r_j+1\right)$ – розмір маски фільтра.

Актуальним є, на прикладі реалізації даної операції, провести дослідження швидкодії її виконання, з урахуванням можливостей оптимізації коду та вибору математичного забезпечення.

Аналіз публікацій та постановка задачі. Сучасне програмне забезпечення залежить від якості інструментів, за допомогою яких воно було створене – компіляторів та інтерпретаторів. Від того, на скільки оптимально компілятор векторизує отриманий код, залежить, чи буде програма, за критерієм швидкодії, задовольняти користувача. В даному контексті – швидкодія залежить від можливостей компілятора векторизувати код програми та від апаратної складової.

На сьогоднішній день компілятори GNU Compiler Collection (далі GCC/G++) [2] та Clang [3] є незамінними інструментами будь-якого розробника. Це заслуга багаторічної праці як відкритого співтовариства FSF (Free Software Foundation) [4], так і університетської розробки, за великої підтримки фірми Apple (на початку). Спільнота FSF розробляє і всіляко підтримує GCC, а Apple зробила ставку на Clang як на компілятор, який є базовим для усіх її продуктів.

GCC являє собою набір компіляторів для різних мов програмування, розроблений в рамках проекту GNU. Початок GCC було покладено Річардом Столлманом, який реалізував перший варіант GCC у 1985 році на нестандартному і непереносному діалекті мови Паскаль; пізніше компілятор був переписаний (Леонардом Тауером і Річардом Столлманом) за допомогою мови С, та випущений у 1987 році як компілятор проекту GNU.

Clang є фронтендом для мов програмування C, C++, Objective-C, Objective-C++ та OpenCL. Для оптимізації (векторизації) і кодогенерації у ньому використовується фреймворк LLVM. Метою проекту є створення заміни GCC. Розробка ведеться згідно з концепцією opensource. У проекті беруть участь працівники декількох корпорацій, у тому числі Google та Apple. Вихідний код доступний на умовах bsd-подібної ліцензії.

Отже, GCC та Clang використовують для отримання оптимального вихідного програмного забезпечення за показником швидкодії виконання. Під оптимальністю розуміється те, що використання команд асемблера SIMD буде на ділянках програми, що представляють найбільш ресурсомісткі/ресурсозатратні частини програмного продукту (ПП), найбільш поширеним. Під ресурсомісткістю слід розуміти те, що дана ділянка програми найбільш часто використовується або містить у собі велику вкладеність циклів, що і призводить до уповільнення загальної програми і т.ін. Таким чином, використовуючи команди SIMD-асемблера, кінцевий ПП отримує приріст продуктивності. Але слід відзначити той автоматизм, з яким компілятори працюють (GCC та Clang): яким би не було завдання, оптимізація йде приблизно за таким алгоритмом перетворення типів даних:

$$\{u8\} \rightarrow \{u16\} \rightarrow \{u32\} \rightarrow \{f32\}[\ldots],$$
(найбільш ресурсномістка частина операцій з f32): […]
$$\{f32\} \rightarrow \{u32\} \rightarrow \{u16\} \rightarrow \{u8\}.$$

Операцію з f32 ([…]) слід розуміти як неефективну (що знижує продуктивність), але таку, що призводить до точності «біт у біт». З цього випливає, що деякі ділянки програмного коду слід переписати вручну і тут вступають у дію фахівці зі знанням мови асемблера (як базового рівня, так і SIMD-команд), які й відповідальні за ухвалення рішення про дотримання принципу «біт у біт», і/або переписування коду (із застосуванням цілочисельного обчислення). У підсумку програма набуває вигляду, який трохи відштовхує середньостатистичного фахівця у сфері програмування, але дає приріст, як мінімум, у 4 рази. Мало відомий факт, що у таких великих

корпораціях, як Google та Apple, є штат програмістів, які розробляють нові чисельні методи, що використовують “single point” арифметику, але результати (за критерієм точності) подібні до обчислень з використанням “floating point” арифметики. Наслідком використання таких алгоритмів є мінімальна припустима втрати точності на противагу до значного підвищення швидкодії алгоритму. Цей ПП роблять за допомогою найновішого асемблера, застосовуючи його найновіші SIMD-команди та, найчастіше, це інлайн асемблер (Inline Assembly).

Тож у подальшому викладенні поставимо за мету дослідити переваги використання оптимізованого коду під час реалізації операції згортки (1) при обробці ЦЗ та отримати нові лінійні оператори для забезпечення виконання саме такої обчислювальної операції.

Виклад основного матеріалу. Десять років тому було домінування CISC-архітектури процесорів. Проте, з розвитком мобільного сегмента ринку (мобільні телефони, планшети, WiFi тощо) першість перейшла до RISC-архітектури процесорів, а точніше – до стандартів процесорів фірми ARM. Найбільш популярними ядрами процесорів з 32-bit архітектурою стали: Cortex-A8, Cortex-A9 та Cortex-A15 [5, с. 13]. Наприкінці 2011 року було опубліковано нову версію 64-bit архитектури ARMv8 з такими стандартами ядер процесорів: Cortex-A53 та Cortex-A57. Найбільш популярним ядром для сегмента одноплатних персональних комп’ютерів (далі – SBC) стала версія Cortex-A53 [6, с. 13]. Прикладом SBC можуть слугувати DragonBoard™ 410c та Raspberry Pi 3 [7].

Векторизація коду дозволяє виконувати, за один такт процесора, безліч одноманітних дій з банком даних (векторним регістром) – SIMD-операцію. На вхід будь-якої SIMD-операції подаються декілька векторних регістрів, довжина яких варіюється залежно від архітектури процесора (CISC або RISC): у CISC довжина векторного регістру може складати 128-bit (xmm) або 256-bit (ymm), 512-bit (zmm); для RISC ця довжина завжди фіксована та складає 128-bit, незалежно від розрядності процесора [5, с. 13; 6, с. 13].

Прикладом, що демонструє якість процедури авто-векторизації програмного забезпечення, з використанням компілятора “GNU C++ compiler” (далі – g++), для архітектур ARMv7-A (прапори компіляції: -O3 -fstrict-aliasing fauto-inc-dec -fprefetch-loop-arrays -mfpu=neon) може бути дослідження її швидкодії на операції (1). Швидкодія програми, що реалізувала операцію (1), досліджувалася на операційній системі Android 5.0.1 з архітектурою процесора Cortex-A53 (процесор: Mediatek MT6752), що дозволяє оцінити як якість авто-векторизації,

так і зворотну сумісність ARMv7-A коду програми з ARMv8 апаратною складовою.

Рисунок 1 – Приклад авто-векторизації операції згортки, отриманий за допомогою програми IDA Pro (1)

Результат авто-векторизації операції (1) представлено на (рис. 1), де подано частину програми мовою асемблера (в основному, SIMD фірми ARM – NEON (32-бітна версія)), яку умовно поділено на 3 секції: 1) перетворення вхідних даних до 32-bit типів (таких як int/float); 2) виконання основного алгоритму; 3) зворотне перетворення та збереження результату (трансляція із 32-bit типів (int/float) у необхідний вихідний тип даних (у наведеному прикладі – це unsigned char (8-bit)).

Отже, як видно з діаграми, для виконання згортки попередньо відбувається перетворення даних до 32-bit типів (рис. 2).

```
VLD1.64         {D18-D19}, [R3@64]!
VMOVL.U8        Q10, D18
VMOVL.U8        Q9, D19
VMOVL.U16       Q13, D20
VMOVL.U16       Q10, D21
VMOVL.U16       Q14, D18
VMOVL.U16       Q9, D19
VMLA.I32        Q8, Q10, Q15
VMLA.I32        Q8, Q14, Q13
VMLA.I32        Q8, Q9, Q12
VADD.I32        D16, D16, D17
VPADD.I32       D22, D16, D16
VMOV.32         R4, D22[0]
```

Рисунок 2 – Приклад автоматичного перетворення компілятором даних до 32-bit типу

Варто також зазначити, що в регістрі Q9 (D18, D19) було передано 16 по 8-bit значень інтесивності кольорової складової растра ЦЗ (ширина векторного регістру в Q9 складає 128-bit = 16 x 8-bit), які були трансльовані у 16 по 32-bit значень та розміщені в регістрах Q9, Q10, Q13, Q14.

Узагальнено, векторізація пройшла таким чином: розгортання внутрішнього циклу, кроком по 4 елемента (та операцій із зображенням/ядром згортки) за одну ітерацію цикла. Це мало б призвести до прискорення операціїї згортки на розмірах ядра, більших за 4 (але не дуже суттево, у порівнянні з експертною векторизацією). Це припущення демонструє рис. 3.

Такий вид авто-векторизації є достатньо поширеним і для перевірки самої авто-векторизації компілятора g++ наведена програма була скомпільована іншим компілятором, що входить у пакет nfk r10e—rc4 (далі – NDK) – LLVM [3]. Результат виявився тим самим, за винятком інших назв регістрів, проте, послідовність дій була еквівалентною (рис.1).

В секції «Виконання згортки» (рис. 1) усі операції відбуваються з 32-bit типами (як i32, так і f32), що дозволяє розробникам ПП, котрі використовують оптимізацію/авто-векторизацію (за допомогою активації прапора -O3 або -Ofast) розраховувати, що вона відбудеться з високою точністю та не призведе до втрати точності розрахунків.

Проте, якщо вхідні дані транслювати в 16-bit діапазон, то, як виявляється, виконання алгоритму операції (1) буде займати, як мінімум, у 4 рази менше команд, а крок зміниться на 32 елемента за один раз – незалежно від розміру ядра згортки. Для підтвердження такого зауваження було проведено серію експериментів, під час яких порівнювалася швидкодія операції (1) для різних розмірів маски фільтра (від 3x3 до 11x11) та різних розмірах ЦЗ (від 10^6 до $8 \cdot 10^6$ пікселів).

Для кожного розміру маски фільтра та для кожного розміру ЦЗ було проведено по 1000 експериментів для порівняння швидкодії при авто-вектризації коду, написаного мовою C++, та при трансляції вручну типів у 16-бітний давпазон мовою асемблера. Усереднене значення T відношення часу виконання:

$$T = \frac{\textit{час виконання при трансляції } 16-bit}{\textit{час виконання при авто-векторизації}}$$

подано на графіку (рис. 3). Як видно, виграш у часі при використанні 16-бітних обчислень складає від 8 до 11 разів у порівнянні з часом виконання операції (1) при авто-векторизації компілятором.

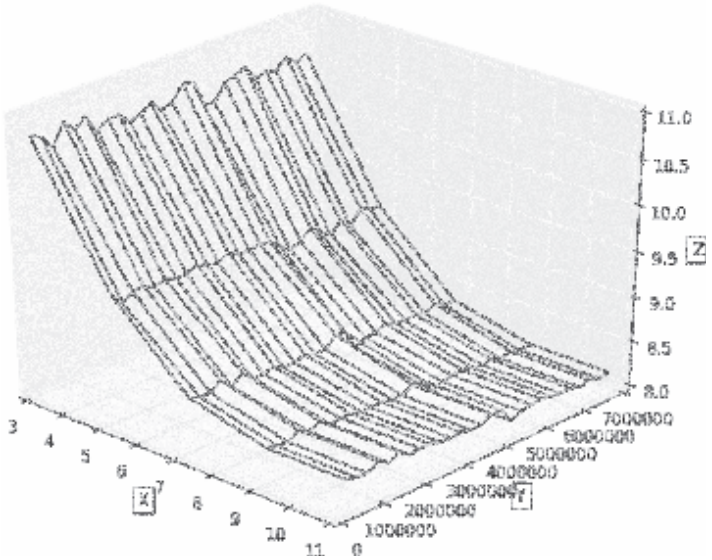

Рисунок 3 – Відношення часу виконання при 16-бітних обчисленнях до часу виконання операції (1) при авто-векторизації: [x] – розмір ядра згортки; [y] – розмір зображення у пікселях; [z] – прискорення

Проте, варто зауважити про існуючі обмеження на використання 16-бітних обчислень при реалізації згортки (1). З урахуванням того, що $p^{\circ}_{ii,jj}$ (значення кольорової складової растра) цілочисельне з діапазону від 0 до 255, можемо записати:

$$\left| \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \gamma_{ii-i,jj-j}\, p^{\circ}_{ii,jj} \right| \le 255 \cdot \sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \left| \gamma_{ii-i,jj-j} \right| .$$

Отже, для забезпечення можливості виконання 16-бітної операції достатньо вимагати, щоб елементи маски γ були типу ShortInt та при реалізації виразу (1) виконувалася нерівність

$$\sum_{ii=i-r_i}^{i+r_i} \sum_{jj=j-r_j}^{j+r_j} \left| \gamma_{ii-i,jj-j} \right| \le 127 . \qquad (2)$$

У подальшому викладенні подамо приклади масок, що задовольняють умові (2). Не зменшуючи загальності, нехай задано деякий безрозмірний растр, кожному пікселю якого поставлено у відповідність двійку індексів $\{(i,j)\}_{i,j\in\square}$, що визначають його місцеположення. Позначимо $\{p_{i,j}\}_{i,j\in\square}$ – для запису обчислювальної схеми при роботі з послідовностями кольорових складових (червоною, зеленою та синьою).

Для реалізації субполосної фільтрації послідовності $\{p_{i,j}\}_{i,j\in\square}$ припускаємо, що значення інтенсивності в кожному пікселі можна подати у вигляді суми

$p_{i,j} = pL_{i,j} + pH_{i,j}$,

де $pL_{i,j}$, $pH_{i,j}$ – відповідно низькочастотні/високочастотні складові, які можна визначати на основі таких лінійних операторів [8]:

$$pL_{i,j} = L\left(p^{i,j}\right) = \sum_{ii=i-1}^{i+1} \sum_{jj=j-1}^{j+1} \gamma L_{ii-i,jj-j}\, p_{ii,jj} \ , \ i,j \in \mathbf{Z} ,$$

$$pH_{i,j} = H\left(p^{i,j}\right) = \sum_{ii=i-1}^{i+1} \sum_{jj=j-1}^{j+1} \gamma H_{ii-i,jj-j}\, p_{ii,jj} \ , \ i,j \in \mathbf{Z} ,$$

де

$$\gamma L = \frac{1}{64}\begin{pmatrix} 1 & 6 & 1 \\ 6 & 36 & 6 \\ 1 & 6 & 1 \end{pmatrix};\ \gamma H = \frac{1}{64}\begin{pmatrix} -1 & -6 & -1 \\ -6 & 28 & -6 \\ -1 & -6 & -1 \end{pmatrix} \qquad (3)$$

або

$$\gamma L = \frac{1}{36}\begin{pmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{pmatrix};\ \gamma H = \frac{1}{36}\begin{pmatrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{pmatrix}. \qquad (4)$$

Зауважимо, що тут і далі в роботі виконання умови (2) вимагається для цілочисельних елементів масок згорток (3) – (4) до ділення на величину, винесену в знаменники.

Для контрастування зображень можна використовувати оператор

$$pK_{i,j} = K\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma K_{ii-i,jj-j}\, p_{ii,jj}\ ,\ i, j \in \mathbf{Z},$$

де

$\left\{pK_{i,j}\right\}_{i,j\in\mathbf{Z}}$ – кольорова складова відконтрастованого растра;

$$\gamma K = \frac{1}{16}\begin{pmatrix} 0 & 0 & 2 & 0 & 0 \\ 0 & 3 & -13 & 3 & 0 \\ 2 & -13 & 48 & -13 & 2 \\ 0 & 3 & -13 & 3 & 0 \\ 0 & 0 & 2 & 0 & 0 \end{pmatrix}, \qquad (5)$$

який, до того ж, є псевдозворотним оператором [9] до оператора $L\left(p^{i,j}\right)$ низькочастотної фільтрації з маскою (4).

Стабілізація цифрового зображення, спотвореного мікрорухом камери фіксації (підвищення різкості) [10], можлива за використання такого оператора:

$$pR_{i,j} = R\left(p^{i,j}\right) = \sum_{ii=i-2}^{i+2}\sum_{jj=j-2}^{j+2} \gamma R_{ii-i,jj-j}\, p_{ii,jj}\ ,\ i, j \in \mathbf{Z},$$

де $\left\{pR_{i,j}\right\}_{i,j\in\mathbf{Z}}$ – кольорова складова растра з покращеною різкістю;

$$\gamma R = \frac{1}{42}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -8 & 1 & 0 \\ -1 & -8 & 74 & -8 & -1 \\ 0 & 1 & -8 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}, \qquad (6)$$

або

$$\gamma R = \frac{1}{15}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -4 & 1 & 0 \\ -1 & -4 & 31 & -4 & -1 \\ 0 & 1 & -4 & 1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}. \qquad (7)$$

Зауважимо, що використання масок (6) та (7) забезпечує, відповідно, менше та більше підвищення різкості.

Для чотирикратного рекурентного поповнення кількості членів послідовності $\left\{p_{i,j,0}\right\}_{i,j\in\mathbf{Z}}$ (двократного рекурентного збільшення лінійних розмірів цифрового зображення) [11; 12] достатньо на кожному κ-му ($\kappa = 1,2,\ldots$) кроці рекурсії визначати члени нової послідовності кольорових складових растра $\left\{p_{2i,2j,\kappa}\right\}_{i,j\in\mathbf{Z}}$ на основі лінійних операторів, що основані на даних попереднього кроку рекурсії:

$$p_{2i,2j,\kappa} = A\left(p^{\kappa-1,i,j}\right), \qquad p_{2i+1,2j,\kappa} = B\left(p^{\kappa-1,i,j}\right),$$

$$p_{2i,2j+1,\kappa} = C\left(p^{\kappa-1,i,j}\right), \qquad p_{2i+1,2j+1,\kappa} = D\left(p^{\kappa-1,i,j}\right),$$

$$i, j \in \mathbf{Z},$$

де $A\left(p^{\kappa-1,i,j}\right)$; $B\left(p^{\kappa-1,i,j}\right)$; $C\left(p^{\kappa-1,i,j}\right)$; $D\left(p^{\kappa-1,i,j}\right)$ можна задавати з умов масштабування. Так, якщо є потреба збільшувати зображення зі згладжуванням, то мають місце такі оператори:

$$p_{a,b,\kappa} = \sum_{ii=i-1}^{ii+1}\sum_{jj=j-1}^{j+1} \gamma\Lambda_{ii-i,jj-j}\, p_{ii,jj,\kappa-1},$$

де

$$\Lambda = \begin{cases} A : a = 2i, b = 2j, \\ B : a = 2i+1, b = 2j, \\ C : a = 2i, b = 2j+1, \\ D : a = 2i+1, b = 2j+1; \end{cases} \tag{8}$$

$$\gamma A = \frac{1}{64}\begin{pmatrix} 1 & 6 & 1 \\ 6 & 36 & 6 \\ 1 & 6 & 1 \end{pmatrix}; \qquad \gamma B = \frac{1}{16}\begin{pmatrix} 0 & 0 & 0 \\ 1 & 6 & 1 \\ 1 & 6 & 1 \end{pmatrix};$$

$$\gamma C = \frac{1}{16}\begin{pmatrix} 0 & 1 & 1 \\ 0 & 6 & 6 \\ 0 & 1 & 1 \end{pmatrix}; \qquad \gamma D = \frac{1}{4}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}.$$

Коли немає потреби збільшувати зі згладжуванням, то варто застосовувати такі оператори:

$$p_{a,b,\kappa} = \sum_{ii=i-1}^{ii+1} \sum_{jj=j-1}^{j+1} \gamma\Lambda_{ii-i,jj-j}\, p_{ii,jj,\kappa-1},$$

де Λ – визначається згідно з (6);

$$\gamma A = \frac{1}{50}\begin{pmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ -1 & 2 & 46 & 2 & -1 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{pmatrix}; \ \gamma B = \frac{1}{82}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -7 & 0 & 0 \\ -1 & 2 & 46 & 2 & -1 \\ -1 & 2 & 46 & 2 & -1 \\ 0 & 0 & -7 & 0 & 0 \end{pmatrix};$$

$$\gamma C = \frac{1}{82}\begin{pmatrix} 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 2 & 2 & 0 \\ 0 & -7 & 46 & 46 & -7 \\ 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & -1 & -1 & 0 \end{pmatrix}; \ \gamma D = \frac{1}{20}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 \\ 0 & -1 & 7 & 7 & -1 \\ 0 & -1 & 7 & 7 & -1 \\ 0 & 0 & -1 & -1 & 0 \end{pmatrix}.$$

Для рекурентного двократного збільшення масштабу (двократного зменшення горизонтального та вертикального розмірів) зображення необхідно на кожному κ-му ($\kappa = 1,2,\ldots$) кроці рекурсії чотирикратно зменшувати кількість пікселів, звільняючи у новому κ-му растрі місце з-під трьох пікселів: праворуч, зверху та зверху-навскіс від кожного (i,j)-го пікселя $(\kappa-1)$-го растра. Тобто, якщо $\{p_{i,j,\kappa}\}_{i,j\in\mathbf{Z}}$ – послідовність однієї із кольорових складових κ-го зменшеного растра,

то $p_{i,j,\kappa} = p_{2i,2j,\kappa-1}$, при цьому пам'ять під розміщення величин $p_{2i+1,2j,\kappa-1}$, $p_{2i,2j+1,\kappa-1}$, $p_{2i+1,2j+1,\kappa-1}$ може бути вивільнена. Але, окрім тривіального визначення членів послідовності $\left\{p_{i,j,\kappa}\right\}_{i,j\in\mathbf{Z}}$ згідно з (7), реалізують збільшення масштабу зображення зі згладжуванням, контрастуванням, направленою фільтрацією тощо, залежно від конкретних потреб. У такому разі величини $p_{i,j,\kappa}$ визначаються на підставі деякого лінійного функціоналу:

$$p_{i,j,\kappa} = F\left(p^{\kappa-1,2i,2j}\right) = \sum_{ii=2i-r}^{2i+r}\sum_{jj=2j-r}^{2j+r}\gamma_{ii-i,jj-j}\,p_{ii,jj,\kappa-1}$$

$i, j \in \mathbf{Z}$, $r = 1,2,\ldots$, що побудований на даних попереднього кроку рекурсії, а в якості γ можна використовувати маски (3) – (7).

Висновки. У роботі проведено дослідження реалізації згортки монохромної складової растра цифрового зображення з квадратною маскою лінійного двовимірного фільтра. Експериментально доведено, що при авто-векторизації коду, що реалізує операцію (1), за допомогою компілятора gcc/g++ для архітектури ARMv7-A, відбувається автоматична конвертація типів даних для 32-розрядних обчислень, задля забезпечення гарантованої точності обчислень. Проте, проведені авторами дослідження доводять, що за допомогою лише 16-бітних операцій та трансляції мовою асемблера ресурсномісткої частини алгоритму можна отримувати виграш від 8 до 11 разів по швидкодії при виконанні операції (1). В роботі наведено умову (2), за якою є можливим досягнення такого виграшу, та подано приклади відповідних лінійних операторів, що можуть мати реалізацію при розробці програмного забезпечення щодо обробки цифрових зображень (та відео).

Подальші дослідження мають за мету: проведення оцінки виграшу при застосуванні 16-бітних обчислень для архітектури процесорів ARMv8 (Mediatek MT6752), що реалізує операцію (1), та отримання нових відповідних лінійних операторів, наприклад, для визначення особливостей ЦЗ, масштабного аналізу тощо.

Бібліографічні посилання

1. Flynn M. J. Very high speed computers. // Proceedings of the IEEE: [Vol: 54. Issue: 12]. 1966. P. 1901–1909.

2. Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов [пер. з англ.]. Киев.

2004. 624 с.

3. Lopes B. C., Auler R. Getting Started with LLVM Core Libraries. Kiyiv. 2014. 314 с.

4. Gay J. Free Software, Free Society: Selected Essays of Richard M. Stallman. Kiyiv. 2002. 230 с.

5. ARM® Cortex®-A15 MPCore™ Processor: (технічний довідник з архітектури процесорів серії Cortex-A15 фірми ARM). 2011–2013. С 392. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0438i/DDI0438I_cortex_a15_r4p0_trm.pdf. Назва з екрана.

6. ARM® Cortex®-A53 MPCore Processor: (технічний довідник з архітектури процесорів серії Cortex-A53 фірми ARM). 2013–2014. С. 620. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf. – Назва з екрана.

7. Raspberry Pi 3 Model B: (технічна специфікація що містить опис апаратного забезпечення одноплатного персонального комп'ютера – Raspberry Pi 3 Model B). 2015. С. 1. URL: https://cdn.sparkfun.com/datasheets/Dev/RaspberryPi/2020826.pdf. – Назва з екрана.

8. Приставка П. О. Обчислювальні аспекти застосування поліноміальних сплайнів при побудові фільтрів // Актуальні проблеми автоматизації та інформаційних технологій : зб. наук. праць. Дніпропетровськ. 2006. Т. 10. С. 3–14.

9. Приставка П. О. Побудова контрастних фільтрів за використанням поліноміальних сплайнів // Актуальні проблеми автоматизації та інформаційних технологій : зб. наук. праць. Дніпропетровськ. 2007. Т. 11. С. 15–22.

10. Приставка П. О., Чолишкіна О. Г. Дослідження комбінованих фільтрів для підвищення різкості зображень // Актуальні проблеми автоматизації та інформаційних технологій : зб. наук. праць. Дніпропетровськ. 2009. Т. 13. С. 39–53.

11. Приставка П. О. Поповнення послідовностей відліків функцій двох змінних на основі поліноміальних сплайнів // Вісник НАУ. 2007. № 3–4. С. 36–39.

12. Приставка П. О. Поповнення зі згладжуванням послідовностей відліків функцій двох змінних на основі сплайнів // Математичне моделювання. 2008. № 1(18). С. 9–12.

Надійшла до редколегії 14.11.16.